Systems programming

13 – Synchronization

MEEC LEEC MEAer LEAer MEIC-A

João Nuno Silva

TÉCNICO LISBOA

Bibliography

- Uresh Vahalia, Unix Internals the new frontier, chapter 7
- Sun Micro systems, Multithreaded Programming Guide, chapter 4
-

Mutexes

- Used to guard Critical Regions
- Used to implement mutual exclusion
- Lock unlock
  - Should be done by the same task
- Simple interactions
  - Only one tasks running protected code

Other synchronization

- Multi-threaded programs may require different synchronization
  - Wait inside a critical region!!!!
  - Different resource guarding
  - Different synchronization patterns
  - Just theard execution coordination

Condition variables

Wait inside critical region

- Long critical regions
  - Serializes code
  - don’t take advantage of concurrency
- Wait inside critical regions
  - For a certain condition/event
  - For a a specific variable value
- Impossible to do with regular mutexes

Wait inside critical region

- Thread 1

```
While (1){
    lock(m1)
    n--
    unlock(m1)
}
```

- Thread 2

```
While (1){
    lock(m1)
    if (n == 0)
        n = 100
    unlock(m1)
}
```

Correct
But
active wait

Condition Variables

- Programmers need additional mechanisms to
  - wait inside locked regions
- Without holding the lock while waiting
  - To allow other threads to entering the locked region

- Condition variables make it possible to
  - sleep inside a critical section
- Condition Variables
  - Atomically release the lock & go to sleep
  - Atomically acquire the lock and wake up

Condition Variables

- Each condition variable
  - Consists of a queue of threads
  - Provides three operations
    - Wait();
      - Atomically release the lock and go to sleep
      - Reacquire lock on return
    - Signal();
      - Wake up one waiting thread, if any
    - Broadcast();
      - Wake up all waiting threads
  - The queue contains threads that
    - Are sleeping/blocked
    - Will be waken by the signal/broadcast

Condition Variables

- Condition variable is associated with a mutex
  - That guards the critical region where thread wait
- Wait
  - Releases and acquire corresponding mutex
- Condition variables guarantee that
  - only one thread is inside the critical region

An Example of Using Condition Variables

```
AddToQueue() {
  lock(M1);
  insert_item
  unlock(M1);
}
RemoveFromQueue() {
  while (1)
    lock(M1);
    if(something on queue)
      break
    else
       unlock(m1)
  remove_item
  unlock(M1);
  return item;
}
```

```
AddToQueue() {
  lock(M1);
  insert_item
  Signal(CV1, M1);
  unlock(M1);
}

RemoveFromQueue() {
  lock(M1);
  Wait(CV1, M1);
  remove_item
  unlock(M1);
  return item;
}
```

Condition Variables

- Mutexes implement synchronization by controlling thread access to data
  - If before entering
- Condition variables allow threads to synchronize based upon the actual value of data.
  - If after entering

Condition Variables

- Without condition variables
  - The programmer would need to have threads continually polling to check if the condition is met.
  - Usually in a critical section
- A condition variable
  - Avoids polling (a.k.a. “busy wait”)
  - Useful when a thread needs to wait for a certain condition to be true.
    - Condition modified inside a critical region

Pthreads conditional var

- A condition variable is a pthread_cont_t

```
#include <pthread.h>
pthread_cond_t cond;
```

- Should be initialized before being used

```
int pthread_cond_init(pthread_cond_t *, const
pthread_condattr_t *);
```

  - 1º parâmetro: address of condition variable
  - 2º parâmetro: attributes (can be NULL)
- static initilaization
  - `cont=PTHREAD_COND_INITIALZER;`

Pthreads conditional var

- In pthreads, there are four relevant procedures involving condition variables:
  - pthread_cond_init(pthread_cond_t *cv, NULL);
  - pthread_cond_destroy(pthread_cond_t *cv);
  - pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *lock);
  - pthread_cond_signal(pthread_cond_t *cv);

Conditional Vars - creation/destruction

- Condition variables must be declared with type **pthread_cond_t**, and must be initialized before they can be used.
  - Dynamically
    - pthread_cond_init(cond, attr);
- pthread_cond_destroy(cond)
  - used to free a condition variable that is no longer needed.

pthread_cond_wait

- pthread_cond_wait(cv, lock)
  - is called by a thread when it wants to block and wait for a condition to be true.
- It is assumed that the thread has locked the mutex indicated by the second parameter.
- The thread releases the mutex, and blocks until awakened by a pthread_cond_signal() call from another thread.
- When it is awakened,
  - it waits until it can acquire the mutex,
  - once acquired, it returns from the pthread_cond_wait() call.

pthread_cond_wait

- Waiting on a condition variable:

```
int pthread_cond_wait(
                  pthread_cont_t * cv,
                  pthread_mutex_t * mux);
```

  - wait until other thread signals/broadcasts
  - Guarantees that **mux** is locked after return

•pthread_cond_signal

- pthread_cond_signal()
  - checks to see if there are any threads waiting on the specified condition variable.
  - If not, then it simply returns.
- If there are threads waiting,
  - then one is awakened.
- No assumption about the order in which threads awake
  - It is natural to assume that they will be awakened in the order in which they waited, but that may not be the case…
- Use loop or pthread_cond_broadcast() to awake all waiting threads.

Condition variables

- **pthread_timedwait**
- **pthread_cond_timedwait**
- pthread_mutex_lock();
- locked
- while(condition_is_false)
- pthread_cond_wait();
- locked
- pthread_mutex_unlock();

Signaling waiting thread

- the thread can change the condition variable
  - unlocks at least one thread
    - **int pthread_cond_signal(pthread_cont_t *);**
  - unlocks all threads
    - **int pthread_cond_broadcast(pthread_cont_t *);**
- Other Thread only resumes after this thread releases mutual exclusion

Condition variables

```
void *producer(void *) {
  Produce_data()
  pthread_mutex_lock(&data_mutex);
  insert_data();
  data_avail = 1;
  pthread_cond_signal(&data_cond);

pthread_mutex_unlock(&data_mutex);
}
```

```
void *consumer(void *) {
  pthread_mutex_lock(&data_mutex);
  while( !data_avail ) {
    /* sleep on condition variable*/
    pthread_cond_wait(&data_cond,
                      &data_mutex);
  }
  /* woken up */
  extract_data();
  if (queue is empty)
    data_avail = 0;
  pthread_mutex_unlock(&data_mutex);
}
```

Condition variables

task Producer

```
void *producer(void *) {
    Produce_data()
    pthread_mutex_lock(&data_mutex);


    insert_data();
    data_avail = 1;
    pthread_cond_signal(&data_cond);
    pthread_mutex_unlock(&data_mutex);
}
```

task Consumer 1

```
void *consumer(void *) {
    pthread_mutex_lock(&data_mutex);
    while( !data_avail ) {
        /* sleep on cond var*/
        pthread_cond_wait(&data_cond, &data_mutex);
    }
    extract_data();
    if (queue is empty)
        data_avail = 0;
    pthread_mutex_unlock(&data_mutex);
}
```

Condition variables

- Loop around wait
  - Other threads may be woken up first.
    - It is possible that another thread might acquire the mutex first and change the state
  - Designing for “loose” predicates may be simpler.
    - Define on condition variables that indicate possibility rather than certainty.
    - signaling a condition variable would mean
      - there may be something” for the signaled thread to do, rather than “there is something” to do.
  - Spurious wake-ups can occur.
    - On some implementations, a thread waiting on a condition variable may be woken up even though no other thread actually signaled the condition variable.
    - Due to optimizations and explicitly permitted by SUSv3.

Read-Write Locks

Read-Write Locks

- Critical regions can be relaxed
- Some of the operations on the resource
  - Can be executed concurrently (just read)
  - Others should be serialized (writes)
- Allow concurrent reads speeds program

Read-Write Lock

- Read-write locks permit concurrent reads and exclusive writes to a protected shared resource.
- The read-write lock is a single entity that can be locked in read or write mode.
- To modify a resource, a thread must first acquire the exclusive write lock.
  - An exclusive write lock is not permitted until all read locks have been released.
-
- Read-write locks support concurrent reads of data structures
  - read operation does not change the record's information.
- When the data is to be updated
  - the write operation must acquire an exclusive write lock.

Not fair / Fair scheduling

- Th1 ReadLock (enter)
- Th2 ReadLock (enter)
- Th3 WriteLock (blocked)
- Th2 unlock (leave)
- Th4 ReadLock (enter)
- Th1 unlock (leave)
- Th2 ReadLock (enter)
-
- Th3 is in starvation

- Th1 ReadLock (enter)
- Th2 ReadLock (enter)
- Th3 WriteLock (blocked)
- Th2 unlock (leave)
- Th4 ReadLock (blocked)
- Th1 unlock (leave)
- Th3 enters
- Th2 ReadLock (blocked)
- .

Read-Write Lock

- Initialization
  - int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
  - const pthread_rwlockattr_t *restrict attr);
    - rwlock will contain the reference to the lock
    - attr can be NULL
- Destruction
  - int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
- locking
  - int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  - int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
- unlocking
  - int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);

Sempahores

Mutex

- Mutex is rigid
  - Count only has 2 values – 0 1
  - Order of operations is fixed: lock → unlock
  - Only owner can unlock
- Some problems require relaxation of these properties
  - “Critical regions” with multiple threads
  - One thread unbocking another

Semaphores

- Abstraction for a counter and a task list, used for synchronization purposes.
  - Proposed by Dijsktra in 1965.
  - Do not require active wait.
  - All modern OS include a version of semaphores (Unix, Windows, ...)
- typedef struct{
-     unsigned counter;
-     processList *queue;
- }sem_t;

Semaphores

- S.counter
  - Defines how many tasks can pass the semaphore without blocking
- If s.counter = 0
  - The next task to try to enter will get blocked
- If s.counter is always <= 1
  - Mutual exclusion is guaranteed
  - The length of the queue depends on the number of tasks waiting to enter

Semaphores

- down(S)
  - Used by a tasks when trying to access a resource
    - Access is given by another task issuing up(S).
  - Task can get blocked if capacity is full
    - counter == 0
- up(S)
  - Used by a task to signal the resource availability
  - up(S) is not blocking

Semaphores

- Wait(S), down(S), or P(S)
  - if (S.counter>0)
    - S.counter--;
  - else
    - Block(S); /* insert S into the queue */
- Signal(S), up(S), or V(S)
  - if (S.queue!=NULL)
    - WakeUp(S); /* removes a processfrom queue */
  - else
    - S.counter++;
- Wakeup and block depend on the OS
- P(S) e V(S) come from dutch words prolaag (decrement) and verhoog (incrementar)

Semaphores

- Semaphores allow implementation of other mechanisms
-
- Critical region
  - Initialize a semaphore with counter = 1
  - Do a down when entering the critical region
  - Do a Up when leaving the critical region
- Rendezvous
  - Initialize a semaphore with counter = 0
  - The tasks that does down(S) will get blocked
    - Until other task does the UP(S)
  - Two tasks reandez-vous and continued together

POSIX Semaphores

- A semaphore is an integer whose value is never allowed to fall below zero.

- POSIX includes a set of functions
  - `man sem_overview`
  - `#include <semaphore.h>`
  - `Link program with -lpthread`

POSIX Semaphores

- Two operations can be performed on semaphores:
  - increment the semaphore value by one
    - sem_post()
  - and decrement the semaphore value by one
    - sem_wait()
- If the value of a semaphore is currently zero,
  - then a sem_wait operation will block until the value becomes greater than zero.

POSIX Semaphores

- POSIX offer two forms of semaphores with respect to creation
  - Named semaphores
    - Identified by a global name (null terminated string started with /
    - Multiple processes can operate on the same named semaphore by passing the same name to sem_open
  - Unamed (memory-based semaphores)
    - Created in memory shared by multiple threads

POSIX Semaphores

- POSIZ offers two sharing mechanism
  - Not shared
    - Just threads of the process
  - Shared
    - Unnamed - Shared by threads in related processes parent children or using shared memory
    - Named - Shared by threads in multiple processes
  - Semaphores are also classified by the maximum number of tasks (N) that can access the resources
    - if N equals 1 => binary semaphore

POSIX Semaphores

Named		unnamed
sem_open()		sem_init()
	sem_wait() sem_trywait() sem_post() sem_getvalue()	
sem_close() sem_unlink()		sem_destroy()

Semaphore	POSIX
up	sem_post
down	sem_wait

POSIX unnamed Semaphores

- An unnamed semaphore is a variable of type **sem_t**
  - #include <semaphore.h>
  - sem_t sem;
- Should be initialized before being used by
  - Child processes,
  - threads.
- **`int sem_init(sem_t *sem, int pshared, unsigned int value)`**
  - sem_init() initializes the unnamed semaphore at the address pointed to by **sem**.
  - The **pshared** argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.
  - The **value** argument specifies the initial value for the semaphore.

POSIX unnamed Semaphores

- Unused semaphores should be destroyed
  - int sem_destroy(sem_t *);

```
sem_t semaforo;
if (sem_init(&semaforo,0,1)==-1)
   perror("Falha na inicializacao");


if (sem_destroy(&semaforo)==-1)
   perror("Falha na eliminacao");
```

POSIX named Semaphores

- Allows the synchronization of processes without shared memory
  - Use a global name
    - string following the form **/name**
- Named semaphores are installed/located in /dev/shm, with the name **sem.name**
- A semaphore should be opened before used

POSIX named Semaphores

- sem_t *sem_open(const char *name, int oflag)
- sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)
  - sem_open() creates a new POSIX semaphore or opens an existing semaphore.
  - The semaphore is identified by name.
  - The oflag argument specifies flags that control the operation of the call.
    - If O_CREAT is specified in oflag, then the semaphore is created if it does not already exist.
- If O_CREAT is specified in **oflag**, two additional arguments must be supplied
  - **mode_t**, premitions (owner, group, user)
  - The value argument specifies the initial value for the new sem aphore..
- If **oflag** contains O_CREAT and O_EXCL
    - Error if semaphore already exists
- If **oflag** is O_CREAT and semaphore exists
    - 3rd and 4th parameters are ignored

POSIX named Semaphores

- When the semaphore is of no use should be closed
  - `int sem_close(sem_t *);`
- The last process should
  - Close the sempahore (sem_close)
  - Remove the correponding file
    - `int sem_unlink(const char *);`
- If a process maintains the semaphore opened
  - sem_unlink is blocked until the semaphore is closed
- If the semaphore is not closed/unlinked
  - New uses of the semaphore are undefined....

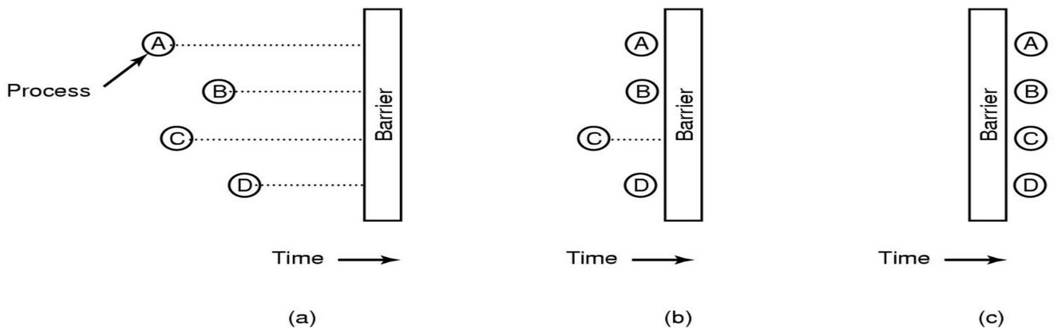
POSIX Semaphores

- down(S) is implemented by the function
  - `int sem_wait(sem_t *);`
  - If counter is zero
    - The thread executing the function is blocked.
      - Remaining thread in the process continue executing
- up(S) is implemenetd by
  - `int sem_post(sem_t *);`

Barriers / rendezvous

Barriers

- Synchronization mechanism that blocks threads until a defined number of threads arrives at the barrier



- Are used when processing is done in steps, that require the completion of a number of threads
  - The next step is only started (by all threads at the same time) after aqll threads have finished the preceding step

Barriers

- Pthreads provides the type pthread_barrier_t

```
pthread_barrier_init(pthread_barrier_t *,
                     pthread_barrierattr_t *,
                     unsigned int)
```

  - 1st parameter - barrier object
  - 2nd parameter - barrier attributes
  - 3rd parameter - number of threads that must call pthread_barrier_wait() before any of them successfully return from the call.

```
pthread_barrier_destroy(pthread_barrier_t *)
```

Barriers

- Wait / Synchronization

`int pthread_barrier_wait(pthread_barrier_t *)`

- All threads block
- When the count value (3rd argument from ini) is reached
  - Count thread unblock and start executing
    - Function returns
- Return value
  - one thread - PTHREAD_BARRIER_SERIAL_THREAD,
  - All others - 0